Accelerating Unstructured Mesh Computations using Custom Streaming Hardware

Kyrylo Tkachov

Supervisor: Professor Paul Kelly

Second marker: Dr. Tony Field.

June 3, 2012

Abstract

In this report we present a methodology for accelerating computations performed on unstructured meshes in the course of a finite volume approach. We implement a custom streaming datapath using Field Programmable Gate Arrays, or FPGAs to perform the bulk of the floating point operations required by the application. In particular, we focus on dealing with irregular memory access patterns that are a consequence of using an unstructured mesh in such a way as to facilitate a streaming model of computation. We describe the partitioning of the mesh and the techniques used to exchange information between neighbouring partitions, using so-called halos. We provide an implementation of a concrete 2D finite volume application and consider the extension to 3D and more complex computations. We evaluate our results by comparing the speedup achieved with analogous GPGPU and multi-core processor implementations.

Acknowledgements

I would like to thank Professor Paul Kelly for giving me so much of his time and ideas and making me aware of scope of the project and the intricacies involved. I would also like to thank Dr. Carlo Bertolli for providing practical advice and explaining the labyrinth that is heterogenous computing. Special thanks go to the team at Maxeler Technologies for helping me out with the details of FPGA-based acceleration and providing support for their excellent toolchain I extend my gratitude to Dr. Tony Field, my personal tutor, who supported me throughout my years at Imperial College and guided so much of my academic development, as well as being the second supervisor on this project.

I would like to thank my mother and grandfather for supporting me through university, both materially and psychologically. Last but not least, I would like to thank my coursemates and friends all over the world, with whom I've had many thought-provoking discussions on every subject imaginable and who always kept me motivated, even when I doubted myself.

Contents

Chapter 1

Introduction

This project presents a methodology for accelerating computations performed on unstructured meshes in the context of Computational Fluid Dynamics (CFD). We use Field Programmable Gate Arrays, or FPGAs, to construct a high-throughput streaming pipeline which is kept filled thanks to an appropriate data layout and partitioning scheme for the mesh. We explore the rearrangement and grouping schemes used to achieve locality of the data points. A formal performance model is constructed to predict the performance characteristics of our architecture and hence justify the design choices made. Appropriate evaluation tests are performed to evaluate the results on a sample CFD application, achieving speedup comparable with state of the art GPGPU and multi-processor solutions. In this section we present a general overview of the problem domain, the hardware platform and the contributions of this project.

1.1 The domain

Computational Fluid Dynamics, or CFD, is a branch of physics focused on numerical algorithms that simulate the movement of fluids and gases and their interactions with surfaces. These simulations are widely used by engineers to design structures and equipment that interact with fluid substances, for example airplane wings and turbines, water and oil pipelines etc.

The required calculations are usually expressed as systems of partial differential equations, the Navier-Stokes equations or the Euler equations, which are discretized using any of a number of techniques. The technique used by our sample application, Airfoil, is the finite volume method that calculates values at discrete places in a mesh and relies on the observation that the fluxes entering a volume are equal to the fluxes leaving it. This project is not concerned with the exact mathematical formulation of these techniques, but they provide a feel for the origins of the problem domain.

1.2 The Airfoil program

The sample program we examine is called Airfoil, a 2D unstructured mesh finite volume simulation of fluid motion around an airplane wing (which has the shape of an airfoil). Airfoil was written as a representative of the class of programs that are tackled by OP2, a framework partially developed and maintained by the Software Performance Optimisation group at Imperial College to abstract the acceleration of unstructured mesh computations on a wide variety of hardware backends.

Airfoil defines an unstructured mesh through sets of nodes, edges and cells and associating them through mappings. Airfoil is written in the C language and these sets are represented at the lowest level as C-arrays. Then data is associated with these sets, such as node coordinates, temperature, pressure etc. The mesh solution is then expressed as the conceptually parallel application of computational kernels on the data associated with each element of a particular set (nodes, edges, cells). These kernels are usually floating point- intensive operations and update the datasets. The procedure is repeated through multiple iterations as desired until a steady-state solution is reached. A more detailed discussion of the unstructured mesh is presented in the Background section of this report.

1.3 FPGAs, streaming and acceleration

In this project we explore the acceleration possibilities of problems in the described domain by using Field Programmable Gate Arrays, or FPGAs. FPGAs are integrated circuits that can be reconfigured on the fly to implement in hardware any logic design. Thanks to this property they provide the development flexibility of software with the benefits of an explicit custom hardware datapath. At a high level, FPGAs can be viewed as a two-dimensional grid of logic elements that can be interconnected in any desirable way.

The FPGA acceleration approach we look at is the streaming model of comutation. In a streaming approach we create a dataflow graph out of simple computational nodes that perform a specific operation on pieces of

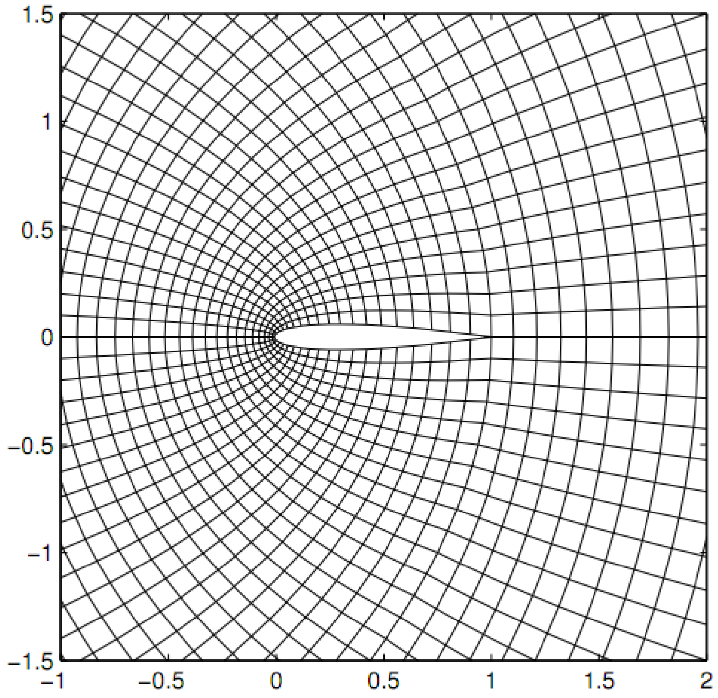


Figure 1.1: Visualisation of a reduced version of the Airfoil mesh

data pushed in and out of them. Connecting these nodes together creates a pipeline through which one can stream an array of data and get one output per cycle thus achieving high throughput. A simple dataflow graph can be seen in figure 1.2. FPGAs are usually programmed using a low level hardware description language like VHDL or Verilog, however many tools have been designed that allow a developer to specify high-level designs. We use MaxCompiler, a compiler that lets us specify the computational graph through a high-level Java API, so we focus on the functional aspects of our design and the tool generates a hardware implementation of it. We use this approach to implement a datapath the kernel described in Airfoil and we then look at approaches to utilise the streaming bandwidth. The FPGAs we consider have a large DRAM storage area attached (>24GB) to them that can be used to store the mesh and utilising the bandwidth of that DRAM fully is key to achieving maximum performance.

During the course of our work it emerges that in order to stream data to and from the accelerator continuously, we need to enforce some spatial locality in the mesh data, thus requiring us to reorder the data and or-

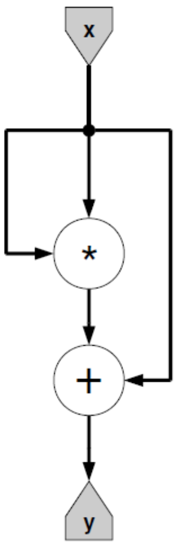


Figure 1.2: A simple dataflow graph that implements the function $y(x) = x^2 + x$.

ganise it into partitions that will be stored in the kernel internally and will need to exchange data with neighbouring partitions through a halo exchange mechanism. This opens a whole new space of decisions that we must make pertaining to the storage layout and streaming responsiblities of the DRAM and the host machine. We present the mesh partitioning schemes that are used to maximise DRAM bandwidth utilisation and maximise pipelining.

We present a performance model that will be used to describe the theoretical performance increase of the system in terms of various parameters like DRAM utilisation, clock frequency etc. Finally we evaluate the performance of our implementation of Airfoil against existing GPGPU and multi-processors cluster implementations.

1.4 Contributions

- We present a methodology for accelerating unstructured mesh computations using deeply pipelined streaming FPGA designs.
- We investigate memory layout issues that arise from efforts to maximise the spatial locality of the mesh.

- We provide a hardware accelerated version of part of the Airfoil program using the methodologies described in this report.
- We provide a predictive performance model that is used to justify our design decisions and provide a formal expression of the potential speedup.
- We investigate the potential for generalisation of the problem and the acceleration of more complex industry-grade unstructured mesh simulations.

Chapter 2

Background

This section provides more detail on the sample application, the representation of meshes, the data sets and the iteration structure of Airfoil. An overview of the Maxeler toolchain is given, which is used to implement the streaming solution we develop. The streaming model of computation is presented in the context of MaxCompiler by walking through steps to build a simple MaxCompiler application. Real number representation is discussed and the concept of a halo is introduced. Previous work in this area is presented and summarised in order to provide a context for the contributions of our work.

2.1 Unstructured meshes and their representation

The spatial domain of the problem can be discretised into either a structured or an unstructured mesh. A structured mesh has the advantage of having a highly regular structure and thus a highly predictable access pattern. If, however, one needs a more detailed solution around a particular area, the mesh would would have to be fine-grained across the whole domain, thus increasing the number of cells, nodes and edges by a large factor even in areas that are not of such great interest. This is where unstructured meshes come in. They explicitly describe the connectivity between the elements and can thus be refined and coarsened around particular areas of interest. This provides much greater flexibility at the expense of losing the regularity of the mesh, forcing us to store the connectivity information that defines its topology. It is useful to have an intimate understanding of the representation of unstructured meshes in order to understand the techniques discussed further on. A graphical example is shown in figure 2.1

In our sample application the mesh distinguishes three main elements: nodes, cells and edges. We have to represent the connectivity information between them. This is done through *indirection maps* that store, for example, the nodes that an edge connects or the nodes that a cell contains. In the application we explore the cells always have four nodes and the edges always connect two nodes and have two cells adjacent. In the more general case of variable-dimension cells (quadrilaterals, triangles, hexagons all mixed together) we would need an additional array storing the indices into the indirection maps and the sizes of the elements. But we do not consider such meshes here.

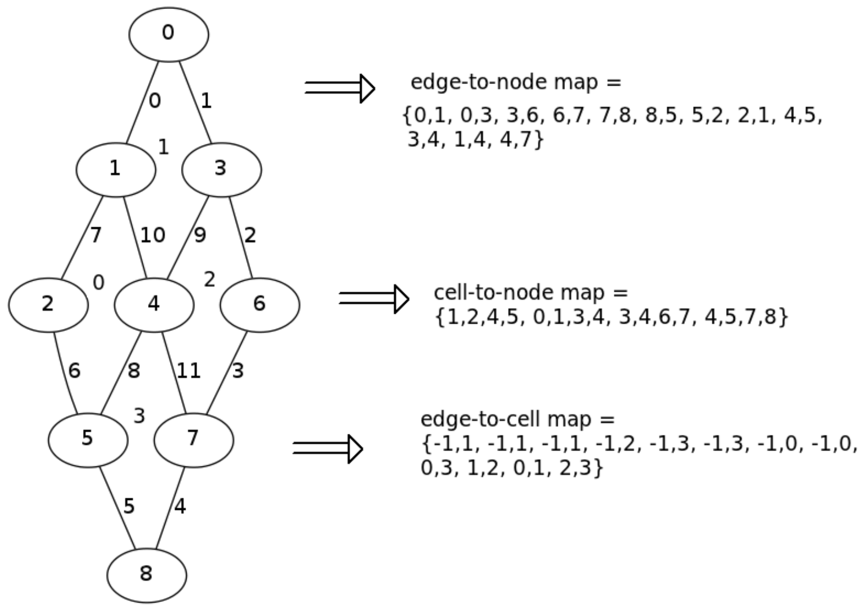


Figure 2.1: An example mesh and its representation using indirection arrays. The cell numbers are shown inside the quadrilaterals formed by the nodes (circles) and edges (edges connecting the nodes). Together with the indirection map, we also store an integer $dim \in \mathbb{N}$ which specifies the dimension of the mapping. Thus, the data associated with element i are stored in the range $[i * dim, \dots, i * (dim + 1) - 1]$ of the relevant indirection map (in the example: the nodes associated with edge 3 are stored at indices $3 * 2 = 6$ and $3 * 2 + 1 = 7$). Note: in the edge-to-cell map -1 represents a boundary cell that may be handled in a special way by a computational kernel.

The above method deals with the connectivity information amongst the different elements of the mesh. The data on which we perform the actual arithmetic calculations is stored in arrays indexed by element number. Such an approach is presented in figure 2.2.

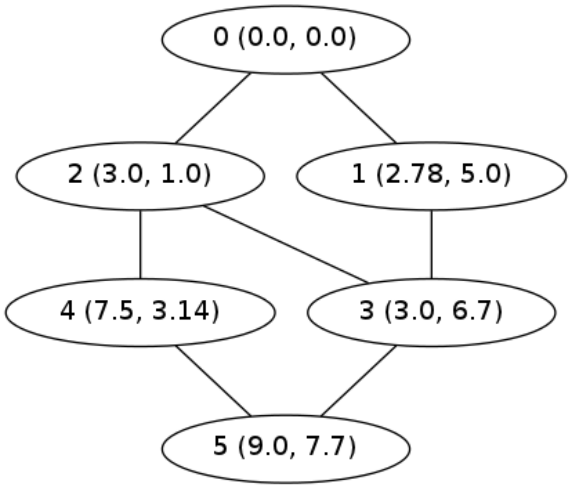


Figure 2.2: An example mesh with coordinate data associated with each node ((x, y) from $node\_id$ (x, y)). The coordinate data will be represented as an array of floating point numbers $x = \{0.0, 0.0, 2.78, 5.0, 3.0, 1.0, 3.0, 6.7, 7.5, 3.14, 9.0, 7.7\}$. Again we also record the dimension of the data (in this case $dim = 2$) in order to access the data set associated with each element. In this example, the coordinate data for node 4 is stored at indices $4 * 2 = 8$ and $4 * 2 + 1 = 9$ of the array x.

2.2 Airfoil

Airfoil was written as a representative of the class of problems we are interested in. It was initially designed as a non-trivial example of the issues tackled by the OP2 framework. Although we are not directly dealing with OP2 in this project, an overview of Airfoil within this context is provided by MB Giles et al [1] because it discusses the acceleration issues arising from the memory access pattern.

The computational work in Airfoil is performed by 5 loops that work one after the other and operate on the nodes, cells and edges of the mesh. They work by applying a kernel on the data item referenced by the node, edge or cell. Conceptually, the application of a kernel to a data item is independent of the application to any other item in the same set, and can therefore be executed in parallel. The complexity comes from reduce operations, where some edges or cells update the same data item (associated with the same node). In these cases care must be taken to ensure the correct update of the data. For parallel architectures such as GPUs and multi-processor clusters this issue can be resolved by enforcing an atomic commit scheme or by colouring the mesh partitions, so that no two partitions update the same data item simultaneously [1].

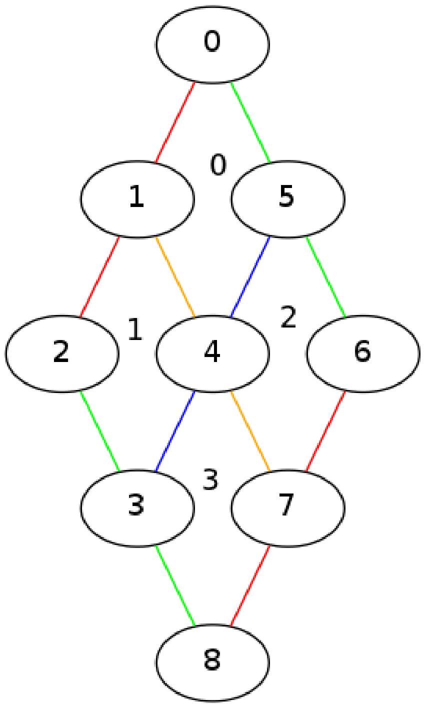


Figure 2.3: An example mesh, showing data dependencies between edges that affect cell data.

Consider figure 2.3. Take for example edges $\alpha = (1, 4)$ and $\beta = (4, 5)$. Say there is a data item x associated with every cell and the processing of an edge increments the data items associated with it's two cells. α and β cannot execute in parallel because they are both associated with cell 0 and can therefore end up using out of date copies of the data associated with cell 0 by the following sequence of events: α reads initial x_0, β reads x_0, α computes $x_\alpha = x_0 + 1$, β computes $x_\beta = x_0 + 1$, α writes back x_α, β

writes back x_β and the final value of x turns out to be $x_\beta = x_0 + 1$ instead of the desired $x_0 + 2$. Some implementations work around this issue by colouring the edges, such that no two edges of the same colour share a cell and can therefore be processed in parallel. Figure 2.3 shows such a colouring. Another option would be to introduce atomic operations and/or locking, but that would approach severely limits parallelisation opportunities.

2.2.1 Computational kernels and data sets

Airfoil defines five computational kernels that iterate over the mesh, performing floating point calculations on the data sets defined over the elements of the mesh. We shall describe them by the elements they iterate over and by the elements they read and modify. As described above, we also define some data sets that are associated with the mesh elements. The datasets defined in Airfoil are shown in table 2.1.

Data set name	Associated with	Type/Dimension	Physical meaning
x	Nodes	$\mathbb{R} \times \mathbb{R}$	Node coordinates
q	Cells	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$	density, momentum, energy per unit volume
q_old	Cells	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$	values of q from previous iteration
res	Cells	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$	residual
adt	Cells	$\mathbb{R}$	Used for calculating area/timestep
bound	Edges	$\{0, 1\}$	Specifies whether an edge is on the boundary of the mesh

Table 2.1: Table showing the data sets and their types. In the actual implementation, we may choose to represent real numbers ($\mathbb{R}$) as standard or double precision floating point numbers or as fixed point numbers (discussed later). Elements of dimension larger than one will be represented as arrays. The physical meaning of these sets is not important, however Airfoil is generally intereseted in computing a steady-state solution for the q data set.

The kernels are presented in table 2.2 along with the datasets they require and modify.

Kernel Name	Iterates over	Reads	Writes
save_soln	Cells	q	q_old
adt_calc	Cells	x, q	adt
res_calc	Edges	x, q, adt	res
bres_calc	(Boundary) Edges	x, q, adt, bound	res
update	Cells	q_old, adt, res	q, res

Table 2.2: Table showing the kernels defined in airfoil and their data requirements.

To show a more concrete example of what these kernels do, the res_calc kernel code in the C language is presented in Listing 1. The rest of the kernels are reproduced in the appendix.

```c
void res_calc(float *x1, float *x2, float *q1, float *q2,
                        float *adt1, float *adt2, float *res1, float *res2) {
  float dx,dy,mu, ri, p1,vol1, p2,vol2, f;
  dx = x1[0] - x2[0];
  dy = x1[1] - x2[1];
  ri = 1.0f/q1[0];
  p1 = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));
  vol1 = ri*(q1[1]*dy - q1[2]*dx);
  ri = 1.0f/q2[0];
  p2 = gm1*(q2[3]-0.5f*ri*(q2[1]*q2[1]+q2[2]*q2[2]));
  vol2 = ri*(q2[1]*dy - q2[2]*dx);
  mu = 0.5f*((*adt1)+(*adt2))*eps;
  f = 0.5f*(vol1* q1[0] + vol2* q2[0] ) + mu*(q1[0]-q2[0]);
  res1[0] += f;
  res2[0] -= f;
  f = 0.5f*(vol1* q1[1] + p1*dy + vol2* q2[1] + p2*dy) + mu*(q1[1]-q2[1]);
  res1[1] += f;
  res2[1] -= f;
  f = 0.5f*(vol1* q1[2] - p1*dx + vol2* q2[2] - p2*dx) + mu*(q1[2]-q2[2]);
  res1[2] += f;
  res2[2] -= f;
  f = 0.5f*(vol1*(q1[3]+p1) + vol2*(q2[3]+p2) ) + mu*(q1[3]-q2[3]);
  res1[3] += f;
  res2[3] -= f;
}
```

Listing 1: Definition of the res_calc kernel with reals represented as single precision floating point numbers. Note the type signature. The kernel requires the element of the dataset x associated with each of the two nodes of the edge we are currently processing and the q, adt and res elements of the two cells associted with the current edge. Note that the res set is updated by incrementing. The important part of this are the data requirements of the kernel and not the exact meaning of the arithmetic operations. The variables *gm*1 and *eps* are global constants that do not need to be passed in explicitly.

2.2.2 Indirection maps

Having defined the data sets and the kernels, we now need to define the indirection maps that express the connectivity of the mesh and the relationships between the elements of the mesh. Airfoil has five such maps called: edge, cell, ecell, bedge, becell. They are presented in figure 2.4.

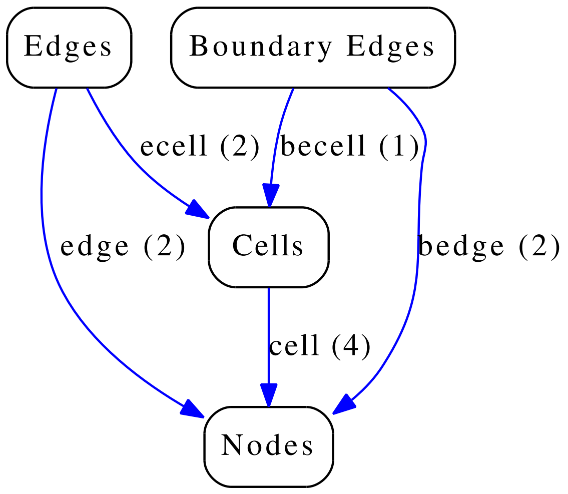


Figure 2.4: Diagram showing the maps between the mesh elements The dimension of the map is shown in parentheses next to the name. Thus the map *edge* relating edges to nodes with dimension 2 means that for each edge, there are two nodes associated with it.

Having specified the data sets, indirection maps and kernels, the application of a kernel on an element is performed by looking up the mesh elements that element is associated with through the indirection maps and using those to access the data sets required by the kernel. For example, the invokation of the res_calc kernel defined in Listing 1 can be done with the following line of C:

```
res_calc(
        &x[2*edge[2*i]], &x[2*edge[2*i+1]], &q[4*ecell[2*i]],
        &q[4*ecell[2*i+1]], &adt[ecell[2*i]], &adt[ecell[2*i+1]],
        &res[4*ecell[2*i]], &res[4*ecell[2*i+1]]
        );
```

Recall that res_calc operates on edges, and correlate the arguments to the type signature in Listing 1. There is a double level of indirection going on here. i is the number of the edge we are currently processing (i ranges in $[0..number\_of\_edges - 1]$). As described in the section on mesh representation, the two nodes corresponding to the edge are stored at indices $2 * i$

and $2 * i + 1$ of the *edge* map. For each of those nodes, res_calc requires the corresponding element in the x data set. Recall from table 2.1 that the x set has a dimension of 2. Therefore the node numbers acquired from $edge[2 * i]$ and $edge[2 * i + 1]$ are multiplied by 2 and used as indices into the array x to access the correct data. Similarly for the rest of the arguments.

The complete iteration step in a sequential implementation of Airfoil is shown in Listing 2. The old values of q are stored in *q_old* and the inner loop runs twice before saving the solution again. The metric *rms* is computed in each iteration that is used to measure the convergence of the solution. The variables *ncell*, *nedge*, *nbedge* represent the number of cells, the number of edges and the number of boundary edges respectively.

A run of the sequential version in Listing 2 on a mesh with 721801 nodes, 1438600 edges, 2800 boundary edges and 720000 cells on an Intel Core2 Duo CPU at 2.8 GHz takes about 183 seconds to complete 1000 iterations. The time spent in each kernel is presented in table 2.3. It is evident that the computation is dominated by the res_calc and adt_calc kernels. In this project we will be concentrating on accelerating the res_calc kernel because it is the most computationally intensive kernel and because it has the most complex data access patterns that make it the interesting case to study. Finding a way to accelerate res_calc would pave the way for accelerating any similar kernel.

Kernel Name	Time spent (seconds)	Percentage of total time (%)
save_soln	6.35	3.47
adt_calc	71.55	39.13
res_calc	81.57	44.62
bres_calc	0.43	0.24
update	22.93	12.54

Table 2.3: Table showing the time spent in each kernel during a run of a single-threaded sequential version of Airfoil. The total run time is 183 seconds.

```
int niter = 1000;
float rms = 0.0;
for(int iter=1; iter<=niter; iter++) {
  for (int i = 0; i < ncell; ++i) {
    save_soln(&q[4*i], &qold[4*i]);
  }
  for(int k=0; k<2; k++) {

    for (int i = 0; i < ncell; ++i) {
      adt_calc(&x[2*cell[4*i]], &x[2*cell[4*i+1]],
               &x[2*cell[4*i+2]], &x[2*cell[4*i+3]],
               &q[4*i], &adt[i]
              );
    }

    for (int i = 0; i < nedge; ++i) {
      res_calc(&x[2*edge[2*i]], &x[2*edge[2*i+1]],
               &q[4*ecell[2*i]], &q[4*ecell[2*i+1]],
               &adt[ecell[2*i]], &adt[ecell[2*i+1]],
               &res[4*ecell[2*i]], &res[4*ecell[2*i+1]]
              );
    }

    for (int i = 0; i < nbedge; ++i) {
      bres_calc(&x[2*bedge[2*i]], &x[2*bedge[2*i+1]],
                &q[4*becell[i]], &adt[becell[i]],
                &res[4*becell[i]], &bound[i]
               );
    }

    rms = 0.0;
    for (int i = 0; i < ncell; ++i) {
      update(&qold[4*i], &q[4*i], &res[4*i], &adt[i], &rms);
    }
  }
  rms = sqrt(rms/(float) ncell);
  if (iter%100 == 0)
    printf(" %d %10.5e \n",iter,rms);
}
```

Listing 2: The iteration structure of Airfoil.

2.3 Hardware platform, Maxeler toolchain and the streaming model of computation

The toolchain we use for implementing the FPGA accelerator is the one developed and maintained by Maxeler Technologies. It consists of the MAX3 cards that contain a Xilinx Virtex-6 chip [3] and up to 48GB of DDR3 DRAM. These cards can be programmed through MaxCompiler[4], which provides a Java-compatible object-oriented API to specify the dataflow graph. MaxCompiler will then schedule the graph, i.e. it will insert buffers that will introduce the appropriate delays in the design that will ensure the correct values will reach the appropriate stages in the pipeline at the correct clock cycle.

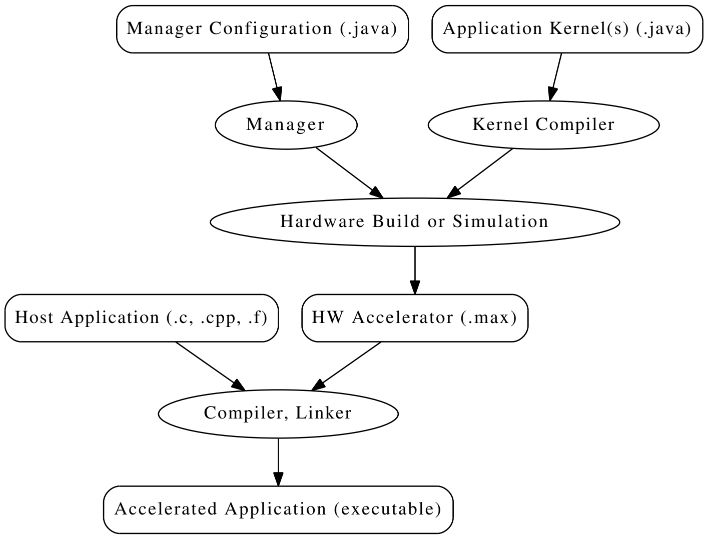

Manager Configuration (.java)
Application Kernel(s) (.java)
Manager
Kernel Compiler
Hardware Build or Simulation
Host Application (.c, .cpp, .f)
HW Accelerator (.max)
Compiler, Linker
Accelerated Application (executable)

Figure 2.5: A diagram of the Maxeler toolchain. The data-flow graphs of the computational kernels are defined using a Java API. A manager connects multiple kernels together and handles the streaming to and from the kernels of data. These are combined by MaxCompiler and compiled into a .max file that can then be linked to a host C/C++ or Fortran application using standard tools (gcc, ld etc).

It will then produce a hardware design in VHDL that will then be further be compiled down to a binary bitstream that configures the FPGA by the Xilinx proprietary tools. The bitstream is then included in what is termed a *maxfile* that contains various other meta-data about the design such as I/O stream names, named memory and register names, various run-time parameters etc. The maxfile can be linked against a normal C/C++ application using standard tools (gcc, ld etc). The interaction with the FPGA is performed by a low-level runtime: MaxCompilerRT and a driver layer: MaxelerOS. A diagram of the toolchain is shown in figure 2.5 [4].

Computational kernels in MaxCompiler have input streams that are pushed through a pipelined dataflow graph and some of them are output from the kernel. Programmatically, a hardware stream is seen as analogous to a variable in conventional programming languages. It's value potentially changes each cycle.

2.3.1 MaxCompiler example

We present a MaxCompiler design that computes a running 3-point average of a stream of floating point values (32 bits) in Listing 3.

```
1  pulic class MovingAverageKernel extends Kernel {
2
3    public MovingAverageKernel(KernelParameters parameters) {
4      super(parameters);
5      HWType flt = hwFloat(8,24);
6      HWVar x = io.input("x", flt ) ;
7      HWVar x_prev = stream.offset(x, -1);
8      HWVar x_next = stream.offset(x, +1);
9      HWVar cnt = control.count.simpleCounter(32, N);
10     HWVar sel_nl = cnt > 0;
11     HWVar sel_nh = cnt < (N-1);
12     HWVar sel_m = sel_nl & sel_nh;
13     HWVar prev = sel_nl ? x_prev : 0;
14     HWVar next = sel_nh ? x_next : 0;
15     HWVar divisor = sel_m ? 3.0 : 2.0;
16     HWVar y = (prev+x+next)/divisor;
17     io.output("y" , y,  flt);
18   }
19 }
```

Listing 3: A MaxCompiler definition of a kernel that computes a moving 3-point average with boundary conditions. Note that the arithmetic operators as well as the ternary if operator have been overloaded for HWVar objects that represent the value of a hardware stream.

MaxCompiler code is written in a Java-like language called MaxJ that provides overloaded operators such as $+, -, *, /$ and ? : . The example in Listing 3 creates a computational kernel that computes a stream of running 3-point averages,named y, from a stream of input values x. The HWVar class is the main representation of the value of a hardware stream at any clock cycle. HWVars always have a HWType that expresses the type of the stream (i.e. an integer, a floating point number, a 1-bit boolean value etc). The $stream.offset(x, -1)$ and $stream.offset(x, +1)$ expressions on lines 7 and 8 extract HWVars for the values of the stream on cycle in the past and one cycle in the future (note that this is internally done by creating implicit buffers, or FIFOs, and scheduling the pipeling accordingly). The ternary if operator ? : creates multiplexers in hardware that express choice. A Java API is provided that contains various useful design elements, such as counters (HWVars that increment their values in many configurable ways every cycle) that can be accessed through the control.count field.

The resulting dataflow graph can be seen in figure 2.6

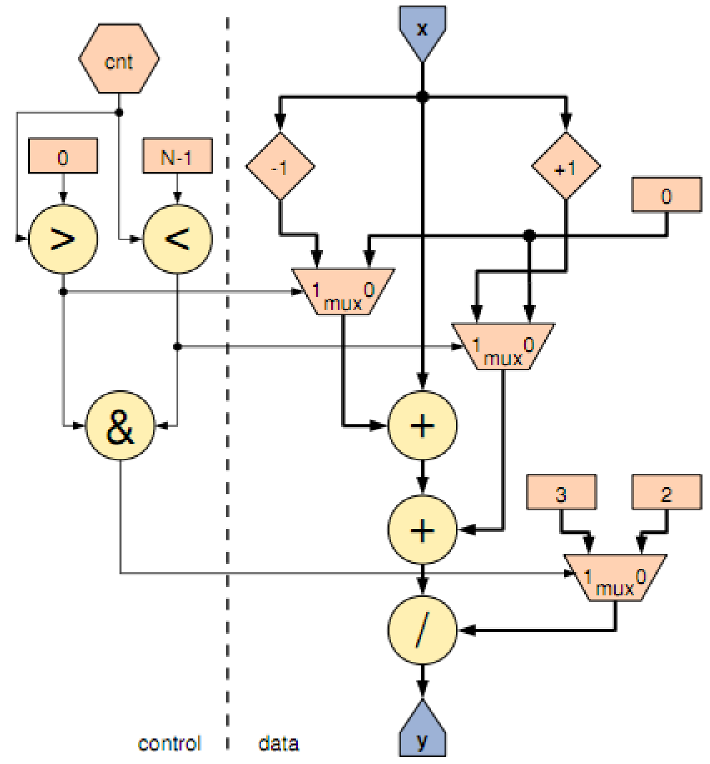


Figure 2.6: The dataflow graph resulting from the code in Listing 3

Kernel designs form part of a MaxCompiler design. The user also specifies a manager that describes the streaming connections between the kernels. A manager can be used to configure a design to stream data to and from the host through PCIe or from the DRAM that is attached to the FPGA. In the manager design, the user will instantiate the kernels and connect them up. Thus for the example in Listing 3 the manager might look like the one in Listing 4.

```java
pulic class MovingAvgManager extends CustomManager {

  public MovingAvgManager(MAXBoardModel board_model,
                          boolean is_simulation, String name) {
    super(is_simulation, board_model, name);
    KernelBlock k
      = addKernel(
          new MovingAverageKernel(makeKernelParameters("MovingAverageKernel"))
                 );

    Stream x = addStreamFromHost("x");
    k.getInput("x") <== x;

    Stream y = addStreamToHost("y");
    y <== k.getOutput("y");
  }
}
```

Listing 4: Manager specification for a MovingAverageKernel that streams the input data ”x” from the host and streams the output data ”y” to the host. The <== operator means connect the right hand side stream to the left hand side stream. The above code instantiates the MovingAverageKernel, creates a stream called ”x” from the host and connects it to the input stream ”x” in the kernel. Then it creates a stream to the host called ”y” and connects to it the output stream ”y” from the kernel.

After we have specified a manger, we can build the design in order to create the .max file using the following lines of code:

```java
public class MovingAvgHWBuilder {
  public static void main(String argv[]) {

    MovingAvgManager m
      = new MovingAvgManager(MAX3BoardModel.MAX3242A,
                             false,
                             "MovingAverage");
    m.build() ;
  }
}
```

This builds our design for a MAX3 card (containing a Xilinx Virtex6 FPGA) using the "MovingAverage" name for the design.

Now that we have a .max file, we can interact with the FPGA from the host code by using the MaxCompilerRT API, an example of which is shown in Listing 5. In order to use the FPGA we must initialise the maxfile as in line 14 and open the device (line 15). The actual streaming to and from the FPGA is done using the max_run vararg function (line 22) where the arrays corresponding to the input data and the allocated space for the output data are specified. The MaxCompilerRT runtime and the MaxelerOS drivers handle the low-level details of PCIe streaming and interrupts.

```c
#include<stdlib.h>
#include<stdint.h>
#include<MaxCompilerRT.h>
#define DATA_SIZE 1024

int main(int argc, char* argv[]) {
        char* device_name = "/dev/maxeler0";
        max_maxfile_t* maxfile;
        max_device_handle_t* device;
        float *data_in, *data_out;

        maxfile = max_maxfile_init_MovingAverage();
        device = max_open_device(maxfile, device_name);

        data_in = (float*)malloc(DATA_SIZE * sizeof(float));
        data_out = (float*)malloc(DATA_SIZE * sizeof(float));

        for (int i = 0; i < DATA_SIZE; ++i) {
                data_in[i] = i;
        }

        max_run(device,
                max_input("x", data_in, DATA_SIZE * sizeof(float)),
                max_output("y", data_out, DATA_SIZE * sizeof(float)),
                max_runfor("MovingAverageKernel", DATA_SIZE),
                max_end());


        for (int i = 0; i < DATA_SIZE; ++i) {
                printf("data_out@%d = %f\n", i, data_out[i]);
        }

        max_close_device(device);
        max_destroy(maxfile);
        return 0;

}
```

Listing 5: A sample host code using the MaxCompilerRT API for the C language.

2.3.2 Hardware

The MAX3 card we use provides 48GB of DRAM that can be accessed with a maximum bandwidth of 38GB/s and a PCIe connection to the host machine that achieves a maximum bandwidth of 2GB/s in both directions. The Virtex6[3] FPGA by Xilinx used in the MAX3 card has about 4MB of fast on-board block RAM that should not be confused with the external DRAM. The host machine can communicate with the card through the PCIe bus using the MaxCompilerRT API. The external DRAM will be used to store the bulk of the mesh data and therefore achieving maximum utilisation of it is be one of the focal points of this project. The top-level parts of the hardware we are dealing with are presented in figure 2.7.

The FPGA provides a number of resources that can be used to specify a design. The Xilinx Virtex6 chip we are using defines four such elements:

- LUTs: LookUp Tables are small elements of combinatorial logic that can be configured to implement any logical function.
- Flip Flops: Stateful elements that can be used as registers to implement accumulators, pipeline stages etc.
- BRAMs: Block RAMs are memory cells that are on the chip itself and can be accessed with very low latency.
- DSPs: Elements custom tuned for fast multiplication.

When building an FPGA design, one must be careful to not use more resources than the chip has to offer, therefore these numbers place a limit on the partition size we can store on the chip at any time, the number of arithmetic pipelines available etc.

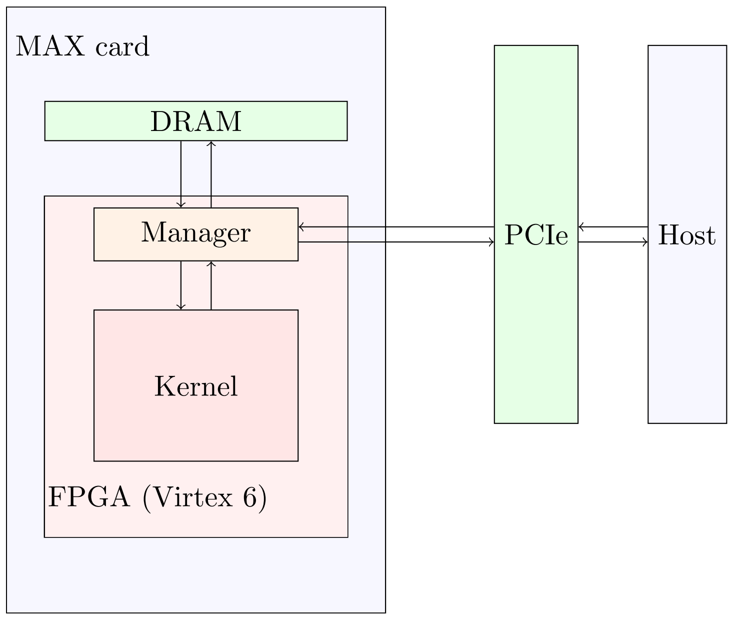


Figure 2.7: Diagram of the hardware parts of a MAX card, showing the relationships between the DRAM, PCIe, the host and the FPGA.

2.3.3 Mesh Partitioning and halos

In computing the optimal memory layout for our application, we have to partition large meshes into partitions that fit in the block RAM of the FPGA. We use a popular and widely available set of tools called METIS developed by George Karypis [5] that uses state of the art techniques to partition meshes, graphs, hypergraphs and matrices according to various parameters like size, edge/hyperedge cut, minimising certain metrics etc. It is a highly robust and efficient tool that we use through its C API. Since an iteration over a mesh element may require data associated with another element, partitions have a set of elements called a *halo region* that consists of all the elements (cells, nodes, edges) that maybe accessed from another partition. Consider figure 2.8. The partitioning is shown with the red line. The four partitions share cells (shown in purple), edges (shown in red) and nodes (along the red line). This presents a difficulty when computing an edge that uses cells in the halo region of another partition, since we are storing a single partition at a time on the device. The method used to deal with this issue is called the *halo exchange mechanism* and it opens up a large design space, with decisions usually dependent on the hardware and communication frameworks.

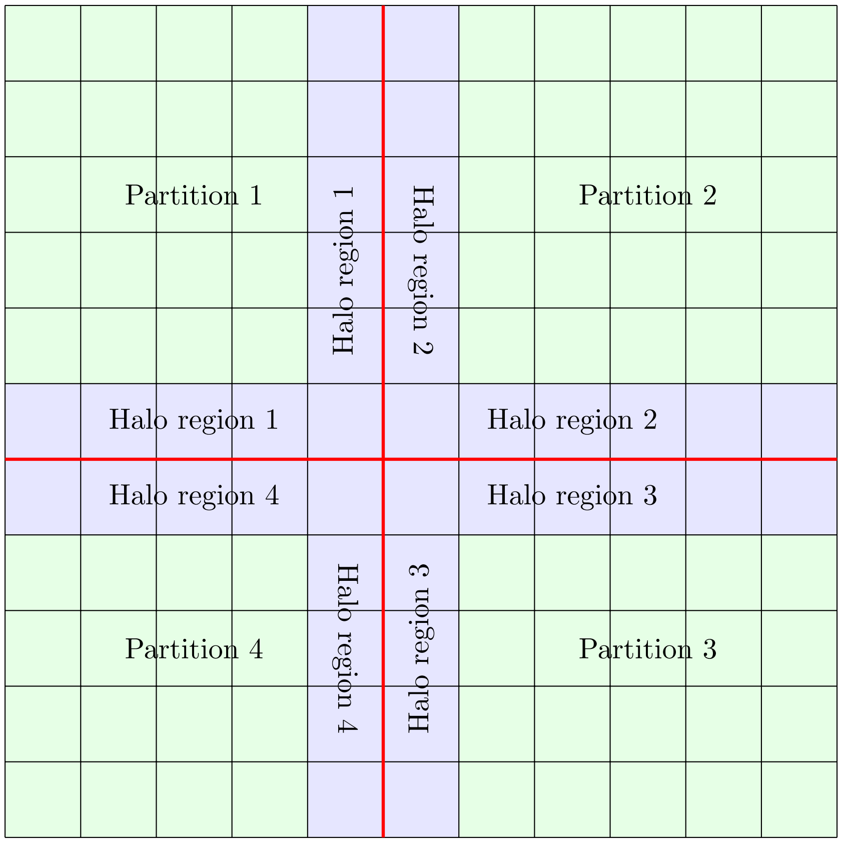


Figure 2.8: A mesh partitioned into 4 partitions, shown in green. The halo regions are shown in purple. Nodes, cells and edges belonging to the halo region can be accessed by another partition.

2.3.4 Floating point vs fixed point arithmetic

As presented in table 2.1, the most important data sets in Airfoil (q, adt, res) consist of real numbers. Therefore a decision must be made on the low level number representation. The most common representation for real numbers in most modern architectures is the IEEE-754[6] floating point representation. This representation stores the number using three fields: the sign bit, the mantissa and the exponent. The representation of a 32-bit floating point number is shown in figure 2.9. The standard also specifies double precision floating point numbers using 64 bits: 11 bits for the exponent and 53 bits for the mantissa. In general, a floating point number with N bits for the exponent can represent a range from $\pm 1 \times 2^{-\frac{2^N}{2}-2}$ to $\pm 2 \times 2^{\frac{2^N}{2}-1}$. For 32-bit single precision numbers, this range is $[\pm 1 \times 2^{-126} .. \pm 2 \times 2^{127}]$. The details of how the mantissa, the exponent and the sign bit are used to encode a real number are presented in the IEEE specification [6]. The decoding of a floating point number involves multiplying the mantissa by 2 raised to the power of the exponent, which is typically an expensive operation. Further details, like exponent bias and normalisation are not discussed here as they are tangential to this section.

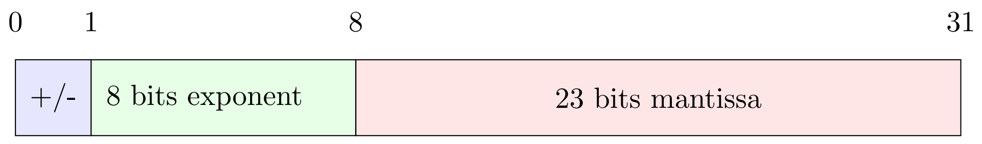


Figure 2.9: The representation of an IEEE-754 single precision floating point number. It has 8 bits for the exponent and 24 bits for the mantissa. However, one bit of the mantissa is used to represent the sign, and is therefore unavailable to the rest of the mantissa.

Since we are working with custom hardware, we have an alternative to floating point numbers for our representation of real arithmetic. We can store a real number as an integer and a fractional part at fixed offsets. This approach makes the arithmetic much simpler and hence faster, but it sacrifices range and accuracy. Fixed point representation is used in applications where the range of the numbers is predictable and not too large, or when the inputs are of limited precision. Faster in this context means fewer cycles taken to perform an operation, which translates to pipeline stages. For example, a fixed point number with 8 integer bits and 8 fraction bits using two's complement mode for negative numbers can repressent numbers from -128 up to $127 + 255/256$ in increments of $1/256$ with each fraction bit representing $1/256$.

2.4 Previous work

Computation using unstructured meshes is widely used in many areas of engineering, not just in fluid dynamics, and there have been many attempts to augument the computation using accelerators. A recent trend has been to use the many cores available on Graphics Processing Units (GPUs) to launch thousands of threads in parallel, exploiting the parallel nature of many of these problems. Other hardware platforms include many-core architectures [1] and large parallel clusters [2]. A project close to this one is OP2 [1]. It is a framework used to specify computations on unstructured meshes in a hardware-agnostic way, allowing the user to concentrate on the functional specification of the algorithm. Airfoil is one of the test programs for that project.

More relevant to this project, there have been attempts to use FPGAs to accelerate such computations. The principles of acceleration using FPGAs are quite different compared to using GPUs or many-core systems. In the case of FPGA acceleration, the performance advantage comes from a custom, application specific, deeply pipelined datapath, often thousands of cycles deep that provides a throughput of one result per cycle. This argument for FPGA acceleration is widely accepted and is the source of the speedups achieved in all current attempts. Given this deep custop pipeline, it is a challenge for the developer to keep the pipeline fully utilised for the maximum amount of time and the techniques used to achieve this have been the main differentiating factors in the applications existing today. The most common approach has been to use on-chip block RAM memory to cache small parts of the mesh and operate on it, cache the results and write them back to main memory.

M.T. Jones and K.Ramachandran [8] formulate the unstructured mesh computation as a sparse matrix problem, $Ax = y$ where A is a large sparse matrix representing the mesh and x is the vector that is being approximated. Their approach uses the conjugate gradient method to iteratively refine the approximation of the x vector. This involves, most importantly, a multiplication of the sparse matrix A with the vector x which forms the bulk of the computation and a subsequent refinement of the mesh and reconstruction of the spares matrix. They formulate the problem as a sparse matrix-vector multiplication, whereas we are interested in iterating computational kernels over the mesh. Furthermore, they are concerned with mesh adaptation and the reconstruction of the sparse matrix in each iteration. We assume a static mesh specified at the beginning of the program.

Morishita et al. [9] examine the acceleration of CFD applications and in particular the use of on-chip block RAM resources to buffer the data in order to keep the arithmetic pipeline as full as possible. This is a more similar approach. However, their approach applies a constant stencil to a grid in 3D and tries to cache points in the grid that will be accessed in the next iteration, thus elimintating redundant accesses to the external memory. This caching/buffering is made possible by the fact that the stencil is of constant shape and thus the memory accesses can be predicted. In our application we have a 2D mesh that does not exhibit this property.

Sanchez-Roman et al. [10] present the acceleration of an airfoil-like unstructured mesh computation using FPGAs. Their solution uses two FPGAs on a single chip that perform different calculations and they identify the need to reason about computation and data access separately. They mention the need to partition larger meshes but they do not discuss techniques for partitioning or the issues arising from data dependencies across partitions. They mention the degradation in performance arising from the unstructured memory access patterns causing cache misses. We present a technique to reorganise the mesh so as to facilitate more well-behaved memory accesses, allowing us to stream data to the datapath efficiently.

In another attempt, Sanchez-Roman et al. [11] recognise the update dependency that occurs during reduction operations, similar to the ones in our res_calc kernel and work around it by adding an accumulator that correctly updates the required data sets. However, their design is used on comparatively small meshes of a maximum of 8000 nodes. Thus all the data can fit into the on-board memory of the FPGA, eliminating the need to consider partitioning issues. The applications we are concerned with usually have meshes of the order of 10^6 edges, which will definitely not fit on the on-chip block RAMs any time soon, thus presenting the need for partitioning.

The existing work gives us many hints towards design decisions. While it was tempting to use fixed point arithmetic for the calculations, Sanchez-Roman et al. [10] hint at the difficulty in porting application from the host side to fixed point and also mention great difficulties that arise in the debugging and verification of the calculations. Furthermore, Durbano et al. [12] find that the error of fixed-point arithmetic accumulates with each iteration in finite difference applications such as the one in Airfoil. They explore the possibility of acceleration of electromagnetics calculations using the Finite Difference Time Domain method (FTDT) which has some similarities to ours from a computational point of view. These factors lead us to discard fixed point representation of real numbers in our solution. All authors mention the relative difficulty of developing FPGA-based systems compared to normal CPU implementations, citing the different mindset required and the inherently more low level reasoning about hardware, architecture and algorithms that must be done to extract the required performance characteristics.

All attempts recognise the need to tame the unstructured memory accesses that arise, and all of them use the on-chip block RAM resources to cache or buffer data before feeding it to the arithmetic pipeline, relying on complex memory access address generators to handle memory accesses. Our approach will also use block RAMs to store parts of the mesh, but we will remove the need for complex memory access patterns by reordering, partitioning and laying out the mesh data in a way that facilitates large, contiguous bursts of streaming. To do that we add a mesh preprocessing stage on the host side that partitions and reorganises the layout in memory of the data.

Some exploration has been done in adapting meshes for particular memory architectures or reordering computations to optimise memory accesses. White B. et al. [13] have explored techniques for reordering computations on CPUs to facilitate efficient memory access, but not from a streaming perspective. There has been some work done in storing meshes for efficient access[14], but it has been focused on block-based CPU and GPU caches. We explore issues that are associated with laying out mesh data for optimal DRAM bandwidth utilisation on FPGAs.

The differentiating factors of this project from existing work are:

- We explore the architectural design space for the accelerator in conjunction with mesh reordering techniques custom-fitted for the chosen architecture.

- Our design aims to work for large meshes, in the order of 10^5 - 10^6 nodes.

- We attempt to keep the design of the hardware accelerator as simple as possible, without complex memory access patterns and relying on host-side preprocessing to figure out the optimal data layout. We are focused on maximising DRAM bandwidth utilisation.

Chapter 3

Design

In this section we discuss the design of the hardware accelerated version of the res_calc kernel from Airfoil. We present the architecture for the FPGA-based accelerator and develop a formal performance model that is used to justify the viability of the architecture. From that point we decide on an optimal data layout that will lead to an implementation of the mesh pre-processing on the host.

3.1 DRAM and mesh storage

The MAX3 cards we have available have a large DRAM memory attached to them, and it is a natural candidate for storing mesh data. The computational kernel on the FPGA can access this memory at a maximum bandwidth of 38GB/s. This bandwidth, however, is only achievable when the memory is accessed in large bursts of contiguous addresses. Random access to this memory, while possible, is very inefficient and wasteful (because the DRAM controller will still read a large chunk of data, but return only the small fraction requested). Because of the representation of the unstructured mesh through indirection maps, processing an edge in res_calc requires a lookup in the *edge* and *ecell* maps, and then a lookup into the x, q and *adt* data sets. If we perform such a two-level dereferencing procedure on the DRAM, the performance is expected to degrade to a point where it is not worth considering. The on-chip block RAMs, on the other hand, are designed to be accessed randomly and do not degrade in performance when accessed so. Thus, we want to push the indirection down to the block RAMs. In other words, mesh connectivity information must not affect the DRAM access pattern and be entirely contained in the data layout and addressing of the block RAMs. This is the argument for storing one mesh partitions in the block RAMs, using connectivity information to access it randomly without penalty and feeding the result into an arithmetic pipeline that will perform the floating point calculations. Under this arrangement, the edges are then represented as a vector of addresses into the node and cell RAMs.

3.2 Result accumulation and storage

Remember that res_calc performs an incrementing operation (also known as *reduction* with addition) on the *res* data set for the cells that each edge accesses. Therefore each edge computes only part of the final value of *res* of its cells. Therefore the arithmetic pipeline, upon processing each edge will produce increments that must be correctly summed up for each cell before that cell is output. Thus the result of the arithmetic pipeline must be added to the current value of res for the relevant cells. We use more block RAMs to store the intermediate *res* results, since the access pattern to *res* is the same as the access pattern for the cells.

The architecture diagram of that description is shown in figure 3.1. The node and cell data for the current partition are streamed in from the DRAM and stored in the block RAMs. They are then read in a pattern described by the connectivity between edges, nodes and cells and fed into the pipeline that produces the *res* increments for two cells that must then be added to the currenty values of *res* for those cells. Thus we have an accumulator node between the block RAMs for *res* and the arithmetic pipeline.

The steps required to process a partition become:

1. Read in node and cell data from DRAM and store it locally.
2. Process the partition data and store the *res* data set locally.
3. Write out the resulting *res* set back to DRAM.

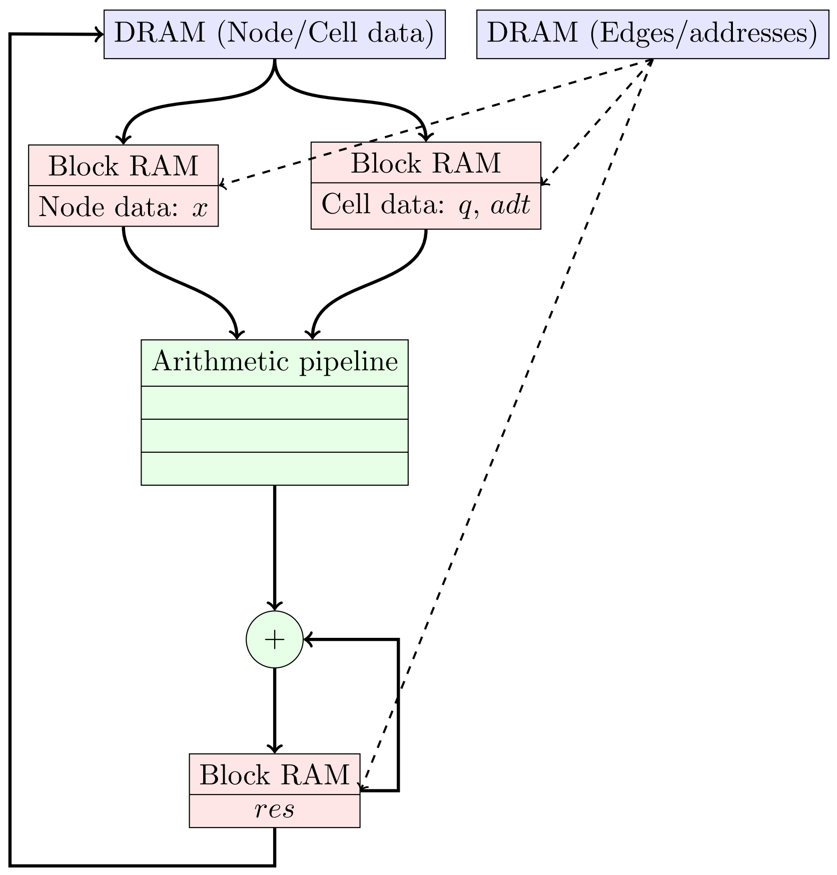


Figure 3.1: Simplified architecture diagram of the accelerator showing the block RAMs storing the node and cell data, the arithmetic pipeline, the result block RAMs and the accumulator. The connectivity information is used to address the block RAMs.

3.3 Halo exchange mechanism

In Airfoil every edge references two nodes and two cells. Those nodes and cells may be contained in the halo region of an adjacent partition. We have to devise a mechanism to acquire the required halo data. Since the mesh connectivity is constant, we can pre-compute the neighbours and the halo regions on the host. The difficulty arises from the reduction operation. A cell that can be accessed from two or more partitions needs to add up the contributions of all its edges, and the four edges that typically reference a cell will not necessarily be in the same partition. We are faced with the problem of updating a cell from two or more partitons. Since we perform the initial partitioning, we can identify these halo cells and store them on the host, not on the DRAM of the card. When processing a partition, we will send these halo cells and nodes to the FPGA via PCIe. The accelerator will then have all the data it needs to process all its edges, however the *res* results that it computes for the halo cells will be only partial results that need to be combined with the results of the other partitions that access those cells. This addition of partial results will be performed on the host. We add some logic to choose the whether to read the cell and node data from the halo RAMs or the normal RAMs to obtain an architecture shown in figure 3.2. This approach to halo exchange is similar to the ghost cell mechanism [7]. Thus the stages for processing a partition become:

1. Read in node and cell data from DRAM. Read in halo node and cell data from PCIe.

2. Process the partition data and store the *res* data set locally. The *res* data for the halo cells is a partial contribution of the final value.

3. Write the resulting *res* set back to DRAM. Send the partial results for halo cells back to the host through PCIe.

4. Once all the partitions are processed, the host adds up the contributions to the halo cells from all partitions.

Remember that the PCIe bandwidth is much lower than that of the DRAM (about 10 times lower), so if the halo region of a partition constitutes a large enough percentage of the total size of the partition, the PCIe transfer becomes dominant and the DRAM will end up being poorly utilised because the kernel will be waiting on the host transfer. This is a factor to keep in mind when choosing partition sizes and partitioning techniques.

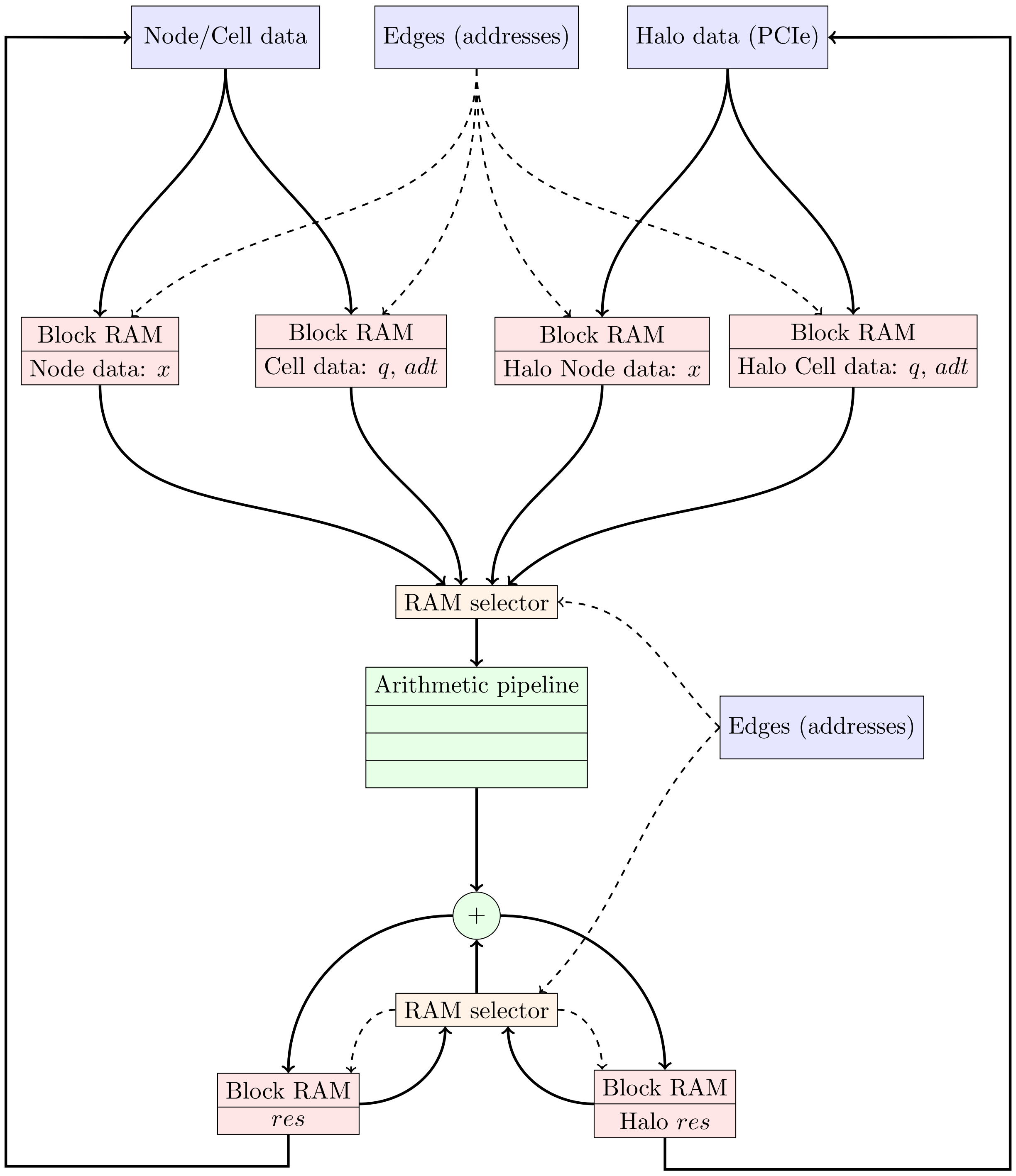


Figure 3.2: Architecture diagram of the accelerator with the PCIe halo exchange mechanism. The RAM selectors will select which RAM to read the cell and node data from based on the edge information. They are also used to pick the RAM to write the results back to. The dashed lines represent addresses that are used to access the RAMs and to determine which RAMs to access.

3.4 Two-level partitioning

The astute reader will notice that in order to process the partition, we first need to stream in the entire partition, process it fully and write it back. During the processing phase we are not streaming anything in or out of the kernel, leaving the DRAM unutilised, thus wasting bandwidth. To mitigate this, we introduce a second level of partitioning on the mesh. Each partition will be split into two *micro-partitions*(μpartitions). The idea is that as soon as we finish reading in the first mircro-partition, we can immediately start processing it while reading in the second micro-partition. Then, when the second mircro-partition has finished streaming in and the processing has finished for the first one, we can write out the results of the first one while processing the second. This allows us to achieve an overlap of data transfer and computation in a scheme reminiscent of a simple processor pipeline, where the fetch, computation and commit stages overlap to increase the utilisation of the relevant units. As shown in figure 3.3, the two micro-partitions will invariably share some elements in a small region we call the *intra-partition halo*(IPH). Care must be taken to not write out the results of the intra-partition halo or overwrite the data in it before both mirco-partitions have finished processing. To avoid confusion we call the top-level partitions macro-partitions. This approach gives us the following phases in the accelerator:

1. Read in non-halo data for first micro-partition plus the intra-partition halo. If this is not the first macro-partition, write out the non-halo data for the second micro-partition and the intra-partition halo.

2. Process first micro-partition, read in the non-halo data for second micro-partition.

3. Process second micro-partition, write out the non-halo data for the first micro-partition.

Note that "read in" and "write out" include both the DRAM and PCIe halo transfers. This approach is expected to greatly improve DRAM utilisation at the expense of slightly more complex control logic. This imposes a constraint on the mesh layout in the DRAM and on the host machine. The first micro-partition should be stored before the intra-partition halo and the second micro-partition last.

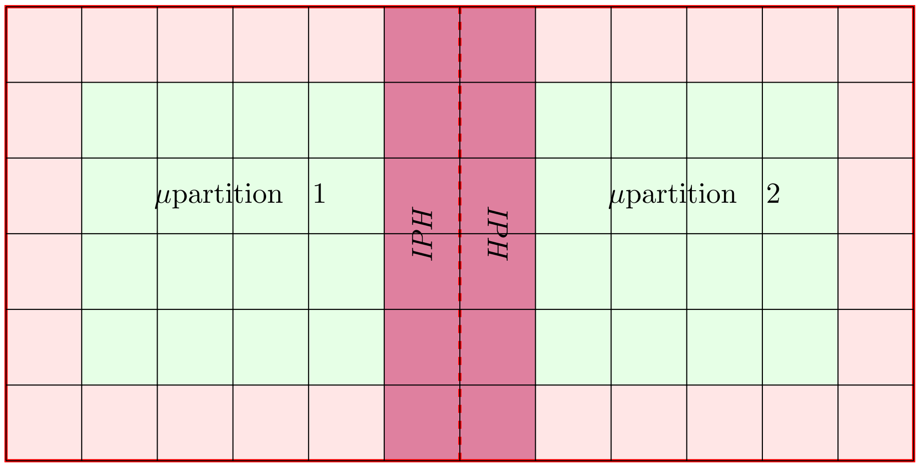


Figure 3.3: Partitioning of a top-level partition into two micro-partitions. This introduces a new intra-partition halo region, shown here in crimson.

The inputs, outputs and RAMs of the kernel can be controlled through enable signals that predicate their function on some boolean condition that we can define. This gives us a straightforward way to control when the kernel reads, processes or writes data. We can define a state machine with internal counters that can be used to keep track of the progress of each phase and signal the I/O units when data needs to be read in or written out. It can also be used to control the block RAMs, specifying when to commit the data found on their input ports. For this, the state machine will need to know the sizes of the micro-partitions, the intra-partition halo and the external halo. This can be added to the design as a separate stream that contains vectors of integers that represent the required sizes. Compared to the sizes of the partitions, the size of the size vector is negligible and therefore does not impact the performance of the memory system. Finally, we arrive at the architecture shown in figure 3.4. The interleaving of I/O and processing gives rise to a pipeline-like execution pattern, shown in figure 3.5. Notice how at every stage there is I/O activity that keeps the DRAM and PCIe streams busy.

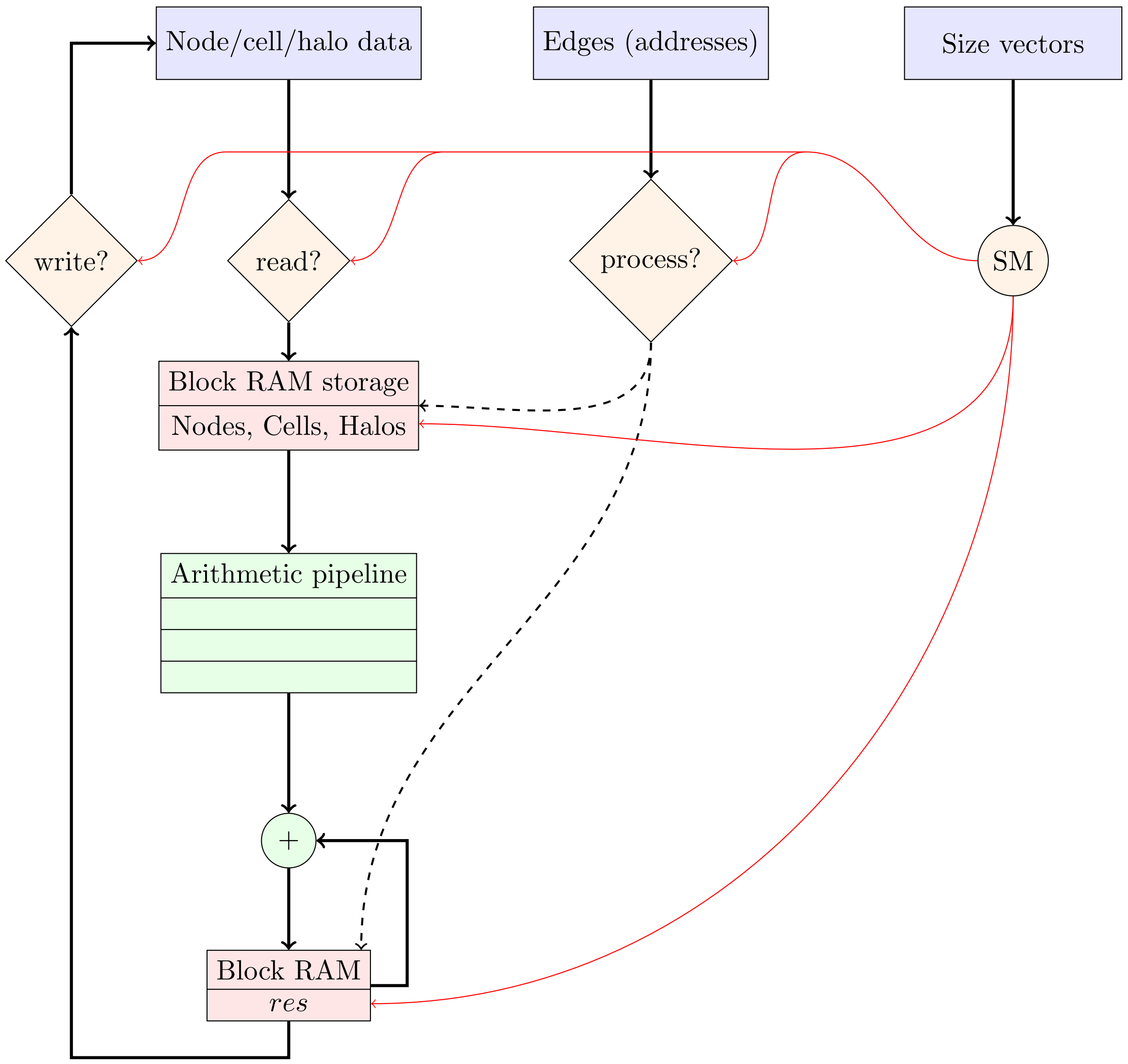


Figure 3.4: Architecture diagram showing the addition of a state machine (node SM) that controls the I/O and the processing. The red wires represent the boolean enable signals. The halo and normal RAMs as well as the RAM selectors are shown in merged blocks for brevity.

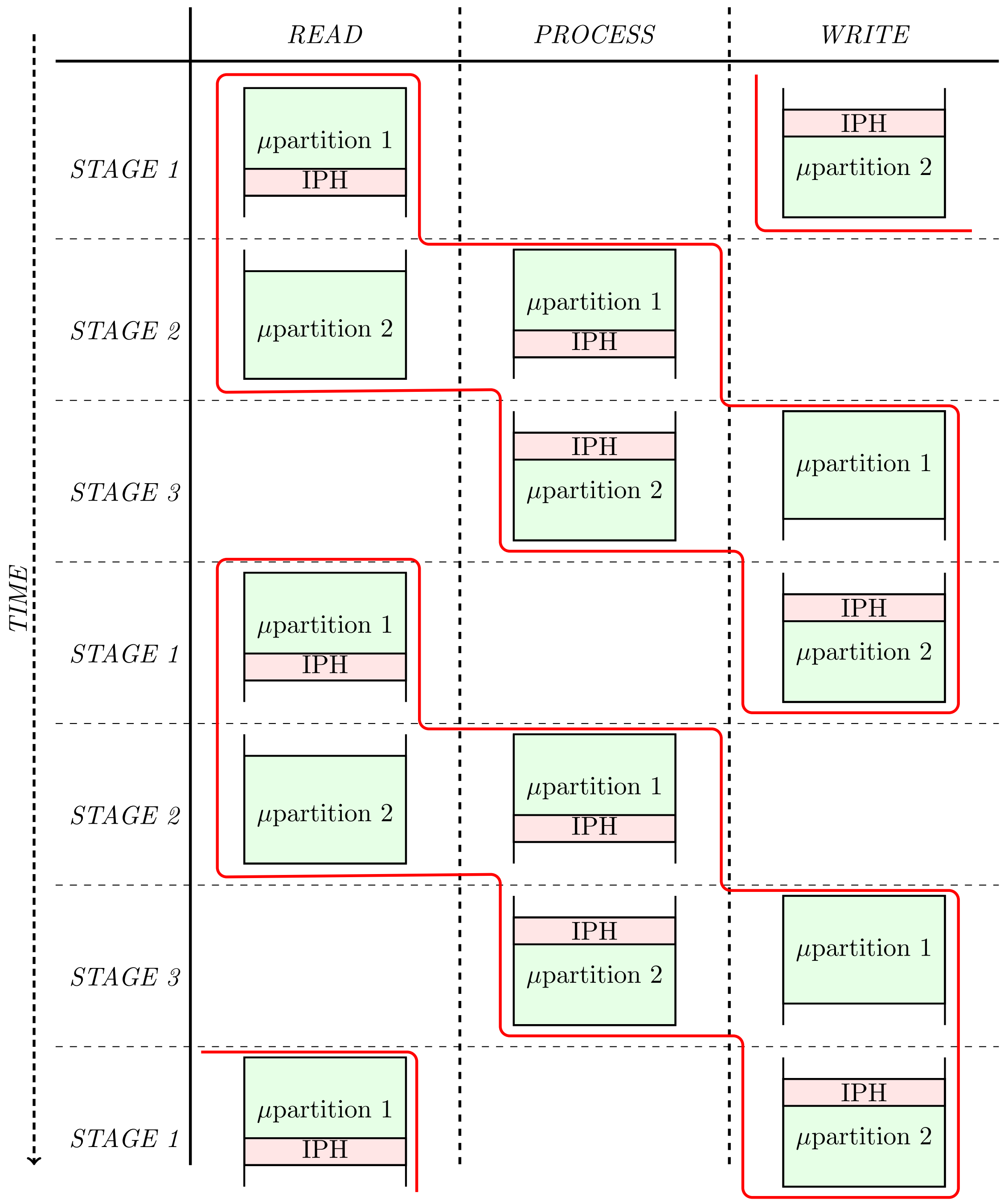


Figure 3.5: Diagram showing the overlapping of execution and I/O thanks to the two-level partitioning scheme. Note that both micropartitions need the intra-partition halo (IPH) in order to be processed, so the IPH can only been written out together with the second micropartition after all of the micropartitions (μpartitions) have been processed. The red boxes represent the progress of a single (macro)partition through the accelerator phases.

3.5 The case for a custom streaming pipeline

The floating point calculations will be performed by the arithmetic pipeline, custom designed for the res_calc kernel. In a general purpose core, like on a CPU and to a somewhat lesser extent a GPU, the floating point calculations will be performed one after another, writing intermediate values to registers and/or cache. The calculations are expressed as a sequence of instructions. In a custom streaming datapath we specify a dataflow graph that the x, q and adt vectors are streamed through, and the res increments come out of the bottom. The advantage of this approach is that we can add registers at every stage of every calculation to create a deep pipeline with high throughput. Pipelining is a well-known processor design technique for increasing functional unit utilisation and throughput. On conventional processors it is used with some care, avoiding very deep pipelines, because the general purpose workload these CPUs are designed for may include arbitrary sequences of instructions that can potentially create various *pipeline hazards* (such as invalidation of an instruction already in a pipeline because a previous branch instruction was taken, a load memory instruction waiting on a write memory instruction etc.). Adding more pipeline stages may increases throughput (results per clock cycle) of instructions, but it also increases their latency (time for a particular instruction to complete) because of the extra registers that are added to store results between the pipeline stages. A pipeline hazard is usually dealt with by stalling the pipeline (waiting for an instruction to complete execution) or flushing it to remove invalid instructions. These measures introduce a performance penalty that is proportional to the depth of the pipeline [16].

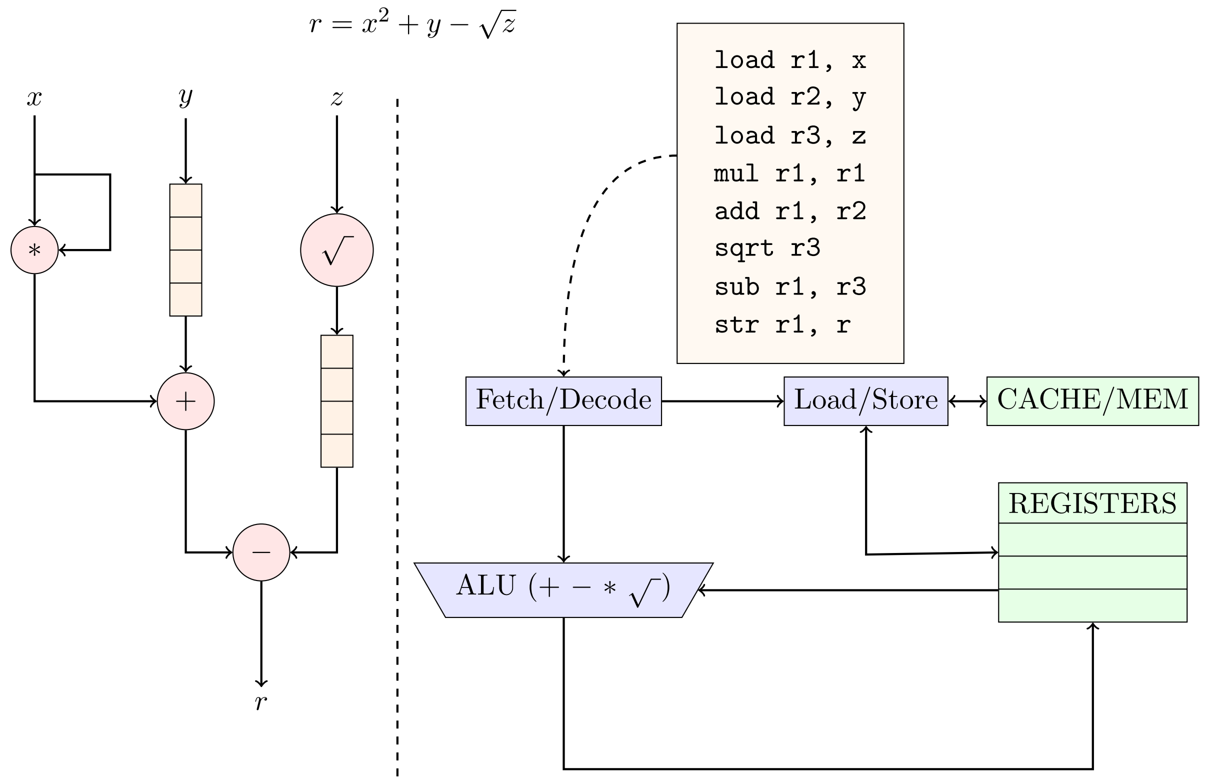
These drawbacks are not applicable to our approach, because we are creating a custom datapath for a known custom workload that will perform specific floating point operations to data that is well-formed for this particular purpose. Since we know beforehand the calculations that will be performed and in what order we can safely pipeline the design as much as possible without worrying about data or control hazards. With that done, the only other concern becomes the task of keeping it occupied for as long as possible as discussed in the sections above.

An important and perhaps counter-intuitive prediction we can make is that because of the extreme pipelining the throughput of the architecture during the processing phase will remain at one result per cycle, regardless of the actual computational workload. As we increase the computational complexity of the kernel, the pipeline gets deeper and therefore takes more cycles to fill in the beginning and flush at the end, but during the time when it's filled (which is most of the time if the architecture works properly and supplies inputs continuously) it produces one result per clock cycle. This gives us an intuition of why this approach to acceleration is a good idea. Assuming large enough data sets, as in Airfoil, the time to fill and flush the pipeline will be amortised by the time spent executing. Compare this observation with the execution of Airfoil on a CPU or a GPU. On those architectures the execution time is expected to increase proportionally to the number of floating point calculations performed. Using a custom streaming datapath, the increase in floating point calculations translates into more resources being used (LUTs, Flip Flops and DSPs), but does not imply a corresponding increase in execution time. This means that a custom, deeply pipelined architecture must win in the asymptotic case against any general purpose architecture as the number of arithmetic operations increases. Of course, in practice, the complexity of the arithmetic pipeline will be limited by the amount of resources available on the chip.

Figure 3.6 shows the difference in the custom streaming approach as opposed to a conventional CPU. Note the absence of a fetch/decode unit in the custom approach, since we are not dealing with instructions. The arithmetic pipeline is designed to implement a particular mathematical function, while on a generic CPU, the Arithmetic and Logic Unit (ALU) can handle arbitrary sequences of arithmetic operations, at the expense of requiring a separate fetch/decode unit as well as a register file to store intermediate results. Notice how the `add r1, r2` instruction cannot execute until the previous instruction `mul r1, r1` has commited its result to register `r1`, creating a potential data hazard that may stall any pipelines present in the functional units of the ALU. In the streaming approach on the left, each of the functional units is pipelined to achieve high throughput. The y stream must be delayed by a FIFO queue before being sent to the adder to compensate for the cycles taken to produce the result from the multiplication unit. Similarly, the output of the square root unit must be delayed/buffered before entering the subtraction node. This way, we can push new values for x, y and z into the pipeline every cycle and after it has been filled, we start receiving one value for r every cycle. Notice, also, that the x^2 and $\sqrt{z}$ functions are computed in parallel, since they operate on different data items. Compare this to the CPU approach on the right, where the computation of a single value for r takes 8 instructions. It is evident that even with sophisticated processor design techniques like out-of-order execution, value forwarding and instruction reordering by the compiler, it is unlikely that the CPU will be able to produce and sustain a throughput of one value per clock cycle since the `load` instructions alone will probably take at least one cycle to fetch the data from the cache or, even worse, the main memory in the event of a cache miss. This should be especially evident on large homogeneous workloads where the time to fill and flush the custom pipeline on the left becomes negligible compared to the time it remains filled, providing maximal throughput, while the CPU case will have to deal with more cache misses that introduce huge performance penalties (in the 1000s of clock cycles) due to the fact that the large data sets will simply not fit into the cache. Of course, the custom streaming approach also demands that data be fed into the pipeline at every cycle, thus requiring additional effort by the developer to format the data accordingly as discussed in previous sections.

A potential GPU implementation will also suffer from these problems, albeit to a lesser degree. While a GPU has hundreds of cores and arithmetic units available to run hundreds or thousands of threads, they are still general purpose and still suffer from the need to decode instructions and access a register file and a cache/memory hierarchy. Adding more complex arithmetic operations will still increase the time to produce a result in a linear fashion, while in a custom streaming datapath the extra work can either be done in parallel by adding extra function nodes and streams or by adding more nodes to the pipeline, increasing the time to fill/flush it, but maintaining the throughput.

Note that the res_calc arithmetic pipeline has too many nodes to be meaningfully reproduced in a diagram here, but it is constructed using the principles discussed in this section.



```
load r1, x
load r2, y
load r3, z
mul r1, r1
add r1, r2
sqrt r3
sub r1, r3
str r1, r
```

Figure 3.6: Diagram showing the computation of the function $r = x^2+y-\sqrt{z}$ by using a custom streaming datapath (left) and using a sequence of instructions in a conventional CPU (right).

Chapter 4

Implementation

*************************UNDER CONSTRUCTION!!!!!***********************************

In this section we present our design, implementation and evaluation plan and provide the details of each stage.

4.1 Plan

We start by proposing various architectures for solving the problem and we propose a formal model for each one that allows us to predict the performance of the architecture. We then implement our scheme in order to provide real-world results and assess the feasibility of the implementation. This model will also allow us to pick the optimal values of the parameters of the application, thus maximizing performance and saving us the effort of using a trial and error approach to fine tune them. After that we proceed with the implementation of our chosen architecture.

During our work we realize that we have to partition the mesh into chunks that we can fit into the on-chip memory (called BRAM or block RAM) for processing. This requires us to think about and deal with data that overlap partitions (for example edges that begin in one partition and end in another). The set of shared data is known as the 'halo' of the partition and various halo exchange schemes exist. We use the ghost cell exchange mechanism, presented in [7].

After we have decided on the various parameters of the design (partition size, number of arithmetic pipelines, streaming responsibilities etc) we implement our design using MaxCompiler to produce a maxfile that we link to the Airfoil executable that we have modified to partition and layout the data in the decided way.

We can then compare our implementation against the theoretical model developed earlier and track down and explain any and all discrepancies. Then we can compare our implementation against existing implementations that use Nvidia's OpenCL CUDA implementation.

Bibliography

[1] MB Giles, GR Mudalige, Z Sharif, G Markall, PHJ Kelly,
Performance Analysis of the OP2 Framework on Many-core Architectures.
ACM SIGMETRICS Performance Evaluation Review, 38(4):9-15, March 2011

[2] G.R Mudalige, MB Giles, C. Bertolli, P.H.J. Kelly,
Predictive Modeling and Analysis of OP2 on DistributedMemory GPU Clusters
PMBS '11 Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems Pages 3-4

[3] Xilinx Inc.
Virtex-6 Family Overview
http://www.xilinx.com/support/documentation/virtex-6.htm

[4] Maxeler Technologies
MaxCompiler White Paper
http://www.maxeler.com/content/briefings/MaxelerWhitePaperMaxCompiler.pdf

[5] G. Karypis, V. Kumar.
A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs
SIAM Journal on Scientific Computing, Vol. 20, No. 1, pp. 359392, 1999.

[6] IEEE
IEEE Std 754-2008
Publication Year: 2008 , Page(s): 1 - 58

[7] F. B. Kjolstad, M Snir
Ghost Cell Pattern

ParaPLoP '10 Proceedings of the 2010 Workshop on Parallel Programming Patterns

[8] M. T. Jones, K. Ramachandran
Unstructured mesh computations on CCMs
Advances in Engineering Software - Special issue on large-scale analysis, design and intelligent synthesis environments Volume 31 Issue 8-9, Aug-Sept. 2000

[9] H. Morishita, Y. Osana, N. Fujita, H. Amano
Exploiting memory hierarchy for a Computational Fluid Dynamics accelerator on FPGAs
ICECE Technology, 2008. FPT 2008. pp 193 - 200

[10] Sanchez-Roman, D.; Sutter, G.; Lopez-Buedo, S.; Gonzalez, I.; Gomez-Arribas, F.J.; Aracil, J.; Palacios, F.;
High-Level Languages and Floating-Point Arithmetic for FPGABased CFD Simulations
Design & Test of Computers, IEEE, 2011, Volume: 28 Issue:4, pp 28 - 37

[11] Sanchez-Roman, D.; Sutter, G.; Lopez-Buedo, S.; Gonzalez, I.; Gomez-Arribas, F.J.; Aracil, A.;
An Euler Solver Accelerator in FPGA for computational fluid dynamics applications
Proceedings of the 2011 VII Southern Conference on Programmable Logic Crdoba, Argentina April 13 15, 2011

[12] Durbano, J.P.; Ortiz, F.E.;
FPGA-based acceleration of the 3D finite-difference time-domain method
12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004. FCCM 2004.

[13] White B., McKee S. ,de Supinski B., Miller B., Quinlan D., Schulz M., Lawrence Livermore National Laboratory
Improving the computational intensity of unstructured mesh applications
ICS '05 Proceedings of the 19th annual international conference on Supercomputing Pages 341 - 350

[14] Sung-Eui, Y., Lindstrom, P.
Mesh Layouts for Block-Based Caches
IEEE Transactions on visualization and computer graphics, Vol. 12, No. 5, September/October 2006

[15] Shirazi, N., Walters, A., Athanas, P.
Quantitative analysis of floating point arithmetic on FPGA based custom computing machines
IEEE Symposium on FPGAs for Custom Computing Machines, 1995. Proceedings.

[16] Hartstein, A. Puzak, Thomas R.
The Optimum Pipeline Depth for a Microprocessor
ISCA '02 Proceedings of the 29th annual international symposium on Computer architecture Pages 7 - 13.

Chapter 5

Appendix

5.1 Airfoil Kernel definitions in C

Even though we focused on accelerating the res_calc kernel, the other kernels are shown here for the sake of completeness.

```
void adt_calc(float *x1,float *x2,float *x3,float *x4,float *q,float *adt){
  float dx,dy, ri,u,v,c;
  ri = 1.0f/q[0];
  u = ri*q[1];
  v = ri*q[2];
  c = sqrt(gam*gm1*(ri*q[3]-0.5f*(u*u+v*v)));
  dx = x2[0] - x1[0];
  dy = x2[1] - x1[1];
  *adt = fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  dx = x3[0] - x2[0];
  dy = x3[1] - x2[1];
  *adt += fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  dx = x4[0] - x3[0];
  dy = x4[1] - x3[1];
  *adt += fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  dx = x1[0] - x4[0];
  dy = x1[1] - x4[1];
  *adt += fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  *adt = (*adt) / cfl;
}

void bres_calc(float *x1, float *x2, float *q1,
                      float *adt1,float *res1, int *bound) {
```

```
  float dx,dy,mu, ri, p1,vol1, p2,vol2, f;
  dx = x1[0] - x2[0];
  dy = x1[1] - x2[1];
  ri = 1.0f/q1[0];
  p1 = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));
  if (*bound==1) {
    res1[1] += + p1*dy;
    res1[2] += - p1*dx;
  }
  else {
    vol1 = ri*(q1[1]*dy - q1[2]*dx);

    ri = 1.0f/qinf[0];
    p2 = gm1*(qinf[3]-0.5f*ri*(qinf[1]*qinf[1]+qinf[2]*qinf[2]));
    vol2 = ri*(qinf[1]*dy - qinf[2]*dx);

    mu = (*adt1)*eps;

    f = 0.5f*(vol1* q1[0] + vol2* qinf[0] ) + mu*(q1[0]-qinf[0]);
    res1[0] += f;
    f = 0.5f*(vol1* q1[1] + p1*dy + vol2* qinf[1] + p2*dy) + mu*(q1[1]-qinf[1]);
    res1[1] += f;
    f = 0.5f*(vol1* q1[2] - p1*dx + vol2* qinf[2] - p2*dx) + mu*(q1[2]-qinf[2]);
    res1[2] += f;
    f = 0.5f*(vol1*(q1[3]+p1) + vol2*(qinf[3]+p2) ) + mu*(q1[3]-qinf[3]);
    res1[3] += f;
  }
}

void res_calc(float *x1, float *x2, float *q1, float *q2,
                     float *adt1,float *adt2,float *res1,float *res2) {

  float dx,dy,mu, ri, p1,vol1, p2,vol2, f;

  dx = x1[0] - x2[0];
  dy = x1[1] - x2[1];

  ri = 1.0f/q1[0];
  p1 = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));
  vol1 = ri*(q1[1]*dy - q1[2]*dx);
  ri = 1.0f/q2[0];
  p2 = gm1*(q2[3]-0.5f*ri*(q2[1]*q2[1]+q2[2]*q2[2]));
  vol2 = ri*(q2[1]*dy - q2[2]*dx);

  mu = 0.5f*((*adt1)+(*adt2))*eps;

  f = 0.5f*(vol1* q1[0] + vol2* q2[0] ) + mu*(q1[0]-q2[0]);
  res1[0] += f;
  res2[0] -= f;
  f = 0.5f*(vol1* q1[1] + p1*dy + vol2* q2[1] + p2*dy) + mu*(q1[1]-q2[1]);
  res1[1] += f;
  res2[1] -= f;
  f = 0.5f*(vol1* q1[2] - p1*dx + vol2* q2[2] - p2*dx) + mu*(q1[2]-q2[2]);
  res1[2] += f;
  res2[2] -= f;
  f = 0.5f*(vol1*(q1[3]+p1) + vol2*(q2[3]+p2) ) + mu*(q1[3]-q2[3]);
  res1[3] += f;
  res2[3] -= f;
}

void save_soln(float *q, float *qold){
  for (int n=0; n<4; n++) qold[n] = q[n];
}

void update(float *qold, float *q, float *res, float *adt, float *rms){
  float del, adti;

  adti = 1.0f/(*adt);

  for (int n=0; n<4; n++) {
    del = adti*res[n];
    q[n] = qold[n] - del;
    res[n] = 0.0f;
    *rms += del*del;
  }
}
```